CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Lec 16: Midterm #1 Review

Midterm Procedure

- Exam is during normal class time. Same time, same place.
- Seating is assigned randomly
  - Wait outside the room until instructed to enter
  - Immediately place all bags/electronics at the front of the room
- At your seat you should have:
  - Writing utensil
  - UB ID card
  - One 8.5x11 cheatsheet (front and back) if desired
    - Summation/Log rules will be provided
  - Water bottle if desired

Content Overview

	Analysis Tools/Techniques	ADTs	Data Structures
Week 2/3	Summations, Asymptotic Analysis, (Unqualified) Runtime Bounds		
Week 3		Sequence	Array, LinkedList
Week 4	Amortized Runtime	List, Set, Bag	ArrayList, LinkedList
Week 5	Recursion, Induction	Set, Bag	ArrayList (sorted), LinkedList

Analysis Tools and Techniques

Recap of Runtime Complexity

Big-Θ — Tight Bound

- Growth functions are in the **same** complexity class
- If f(n) ∈ **Θ**(g(n)) then an algorithm taking f(n) steps is as "exactly" as fast as one that takes g(n) steps.

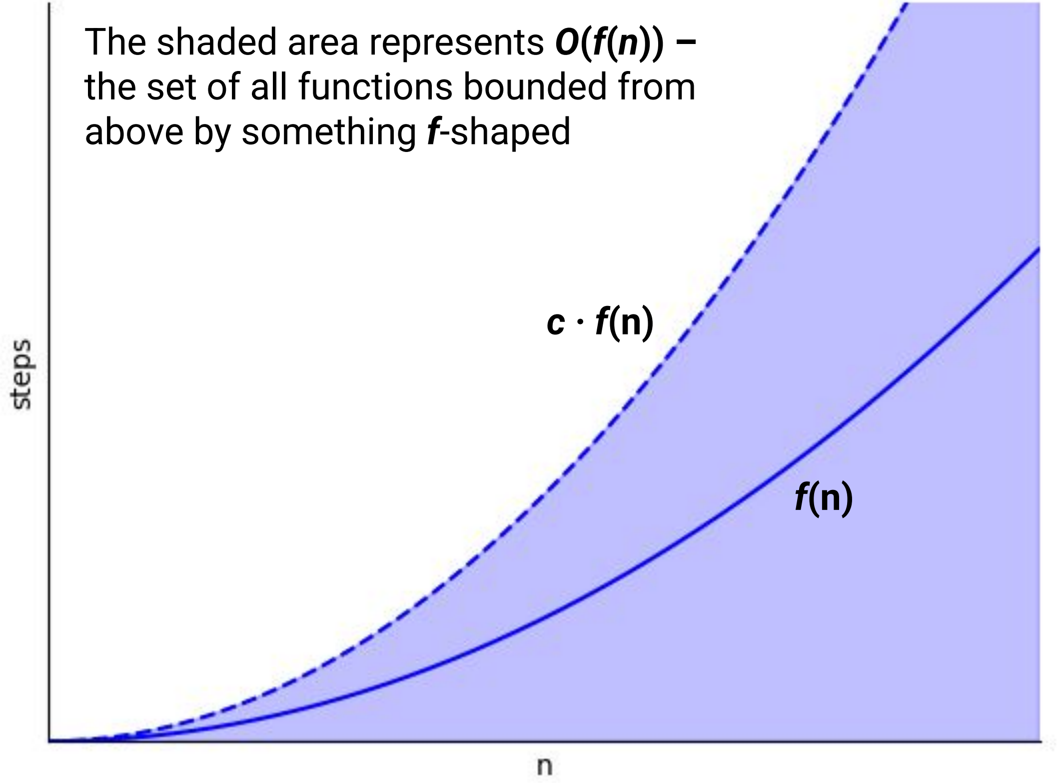
Big-O — Upper Bound

- Growth functions in the **same or smaller** complexity class.
- If f(n) ∈ O(g(n)), then an algorithm that takes f(n) steps is *at least as fast* as one taking g(n) (but it may be even faster).

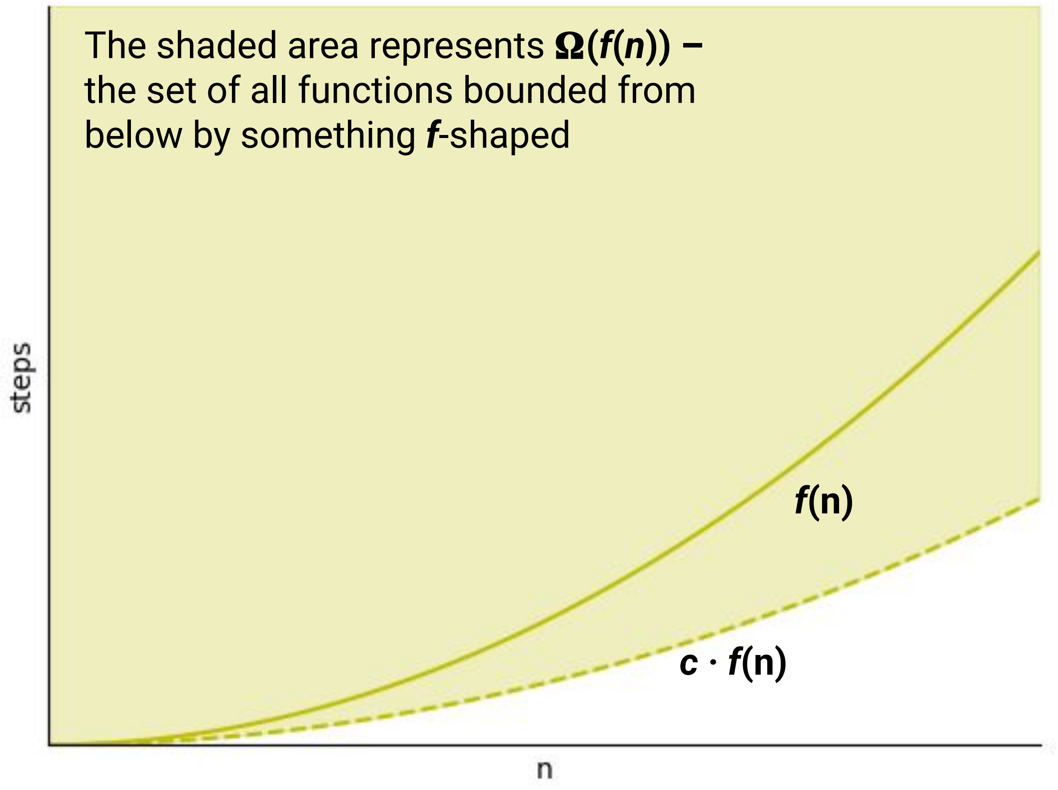
Big-Ω — Lower Bound

- Growth functions in the **same or bigger** complexity class
- If f(n) ∈ **Ω**(g(n)), then an algorithm that takes f(n) steps is *at least as slow* as one that takes g(n) steps (but it may be even slower)

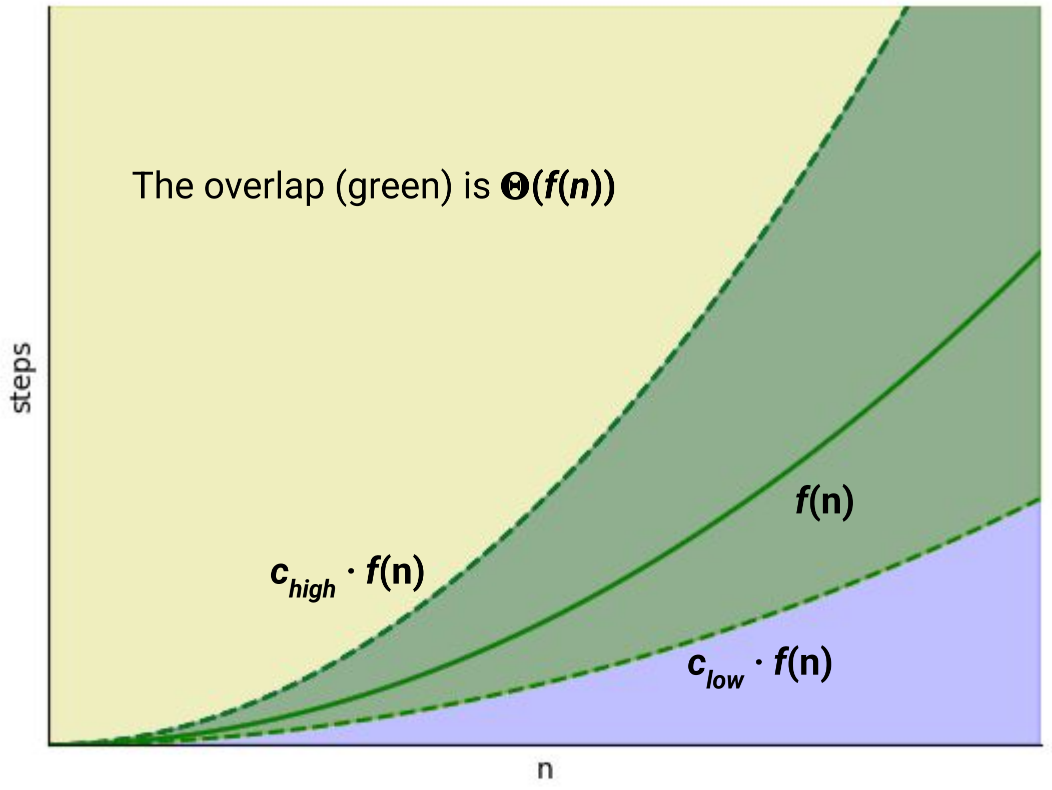
Bounded from Above: Big O



Bounded from Below: Big Ω

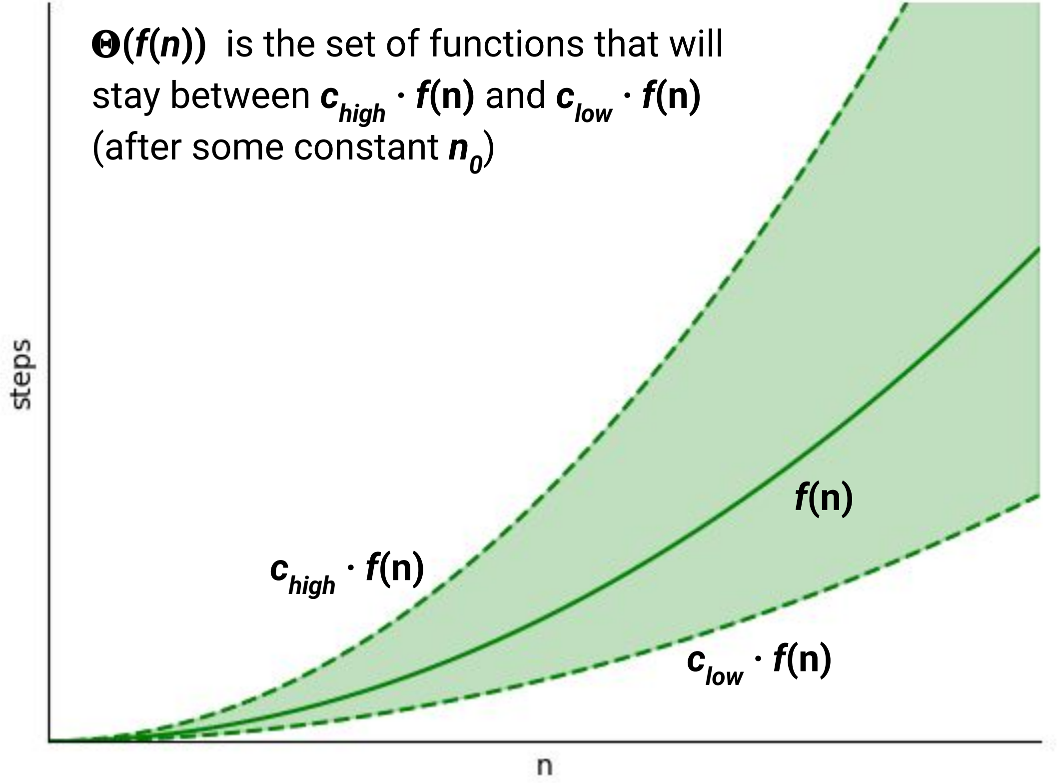


Complexity Class: Big Θ

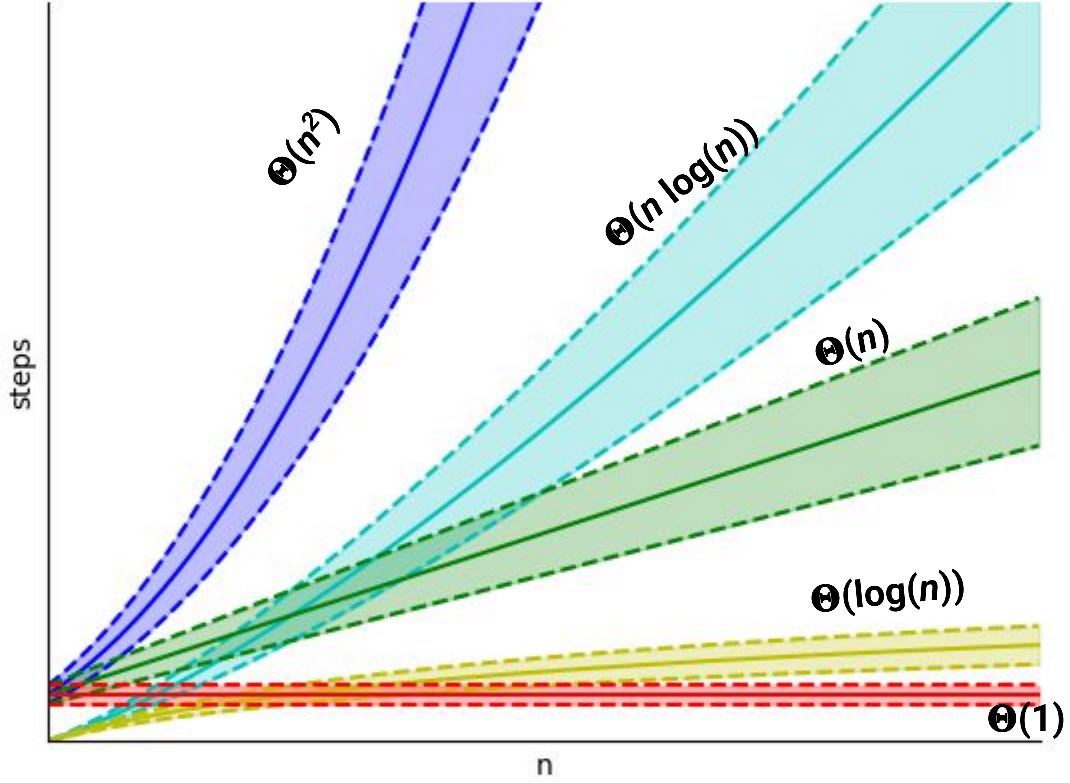


Complexity Class: Big Θ

$\Theta(f(n))$ is the set of functions that will stay between $c_{high} \cdot f(n)$ and $c_{low} \cdot f(n)$ (after some constant n_0)



Complexity Class Ranking



$\Theta(1) < \Theta(\log(n)) < \Theta(n) < \Theta(n \log(n)) < \Theta(n^2) < \Theta(n^3) < \Theta(2^n)$

Common Runtimes (in order of complexity)

Constant Time:	$\Theta(1)$
Logarithmic Time:	$\Theta(\log(n))$
Linear Time:	$\Theta(n)$
Quadratic Time:	$\Theta(n^2)$
Polynomial Time:	$\Theta(n^k)$ for some $k > 0$
Exponential Time:	$\Theta(c^n)$ (for some $c \geq 1$)

Formal Definitions

f(n) ∈ *O*(g(n)) iff exists some constants **c**, $\mathbf{n_0}$ s.t.

  f(n) ≤ c * g(n) for all $\mathbf{n > n_0}$

f(n) ∈ Ω(g(n)) iff exists some constants **c**, $\mathbf{n_0}$ s.t.

  f(n) ≥ c * g(n) for all $\mathbf{n > n_0}$

f(n) ∈ Θ(g(n)) iff **f(n) ∈ *O*(g(n))** and **f(n) ∈ Ω(g(n))**

Shortcut

What complexity class do each of the following belong to:

$f(n) = 4n + n^2 \in \Theta(n^2)$

$g(n) = 2^n + 4n \in \Theta(2^n)$

$h(n) = 100\, n \log(n) + 73n \in \Theta(n \log(n))$

Shortcut: Just consider the complexity of the most dominant term

Multi-class Functions

$$T(n) = \begin{cases} n^2 & \text{if } n \text{ is even} \\ n & \text{if } n \text{ is odd} \end{cases}$$

It is not bounded from above by ***n***, therefore it cannot be in $\boldsymbol{\Theta(n)}$

It is not bounded from below by $\boldsymbol{n^2}$, therefore it cannot be in $\boldsymbol{\Theta(n^2)}$

What is the tight upper bound of this function? $\boldsymbol{T(n) \in O(n^2)}$

What is the tight lower bound of this function? $\boldsymbol{T(n) \in \Omega(n)}$

What is the complexity class of this function? It does not have one!

Amortized Runtime

If n calls to a function take $\Theta(f(n))$...

We say the **Amortized Runtime** is $\Theta(f(n) / n)$

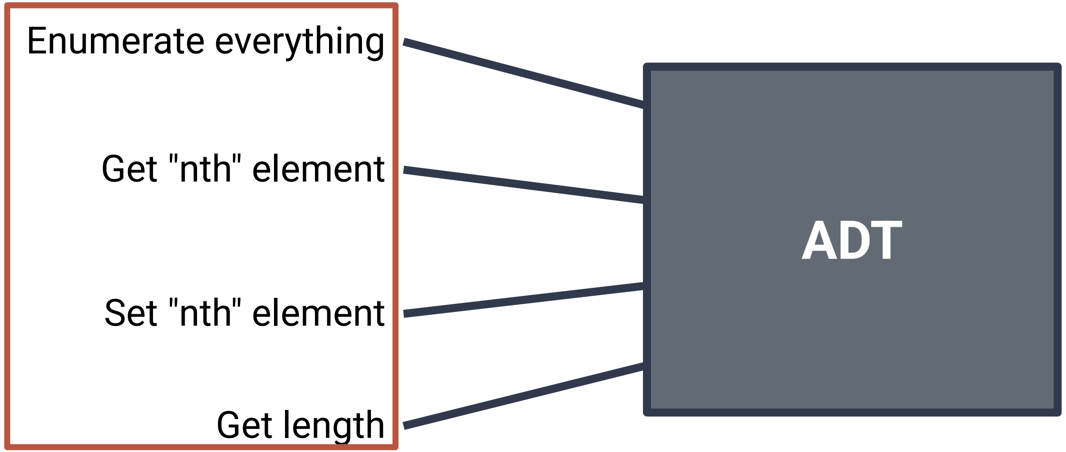
The **amortized runtime** of `add` on an `ArrayList` is: $\Theta(n/n) = \Theta(1)$
The **unqualified runtime** of `add` on an `ArrayList` is: $O(n)$

ADTs and Data Structures

Abstract Data Types (ADTs)

The specification of **what** a data structure can do

What's in the box? …we don't know, and in some sense…we don't care



Usage is governed by **what** we can do, not **how** it is done

Abstract Data Type vs Data Structure

ADT

The interface to a data structure

*Defines **what** the data structure can do*

Many data structures can implement the same ADT

Data Structure

The implementation of one (or more) ADTs

*Defines **how** the different tasks are carried out*

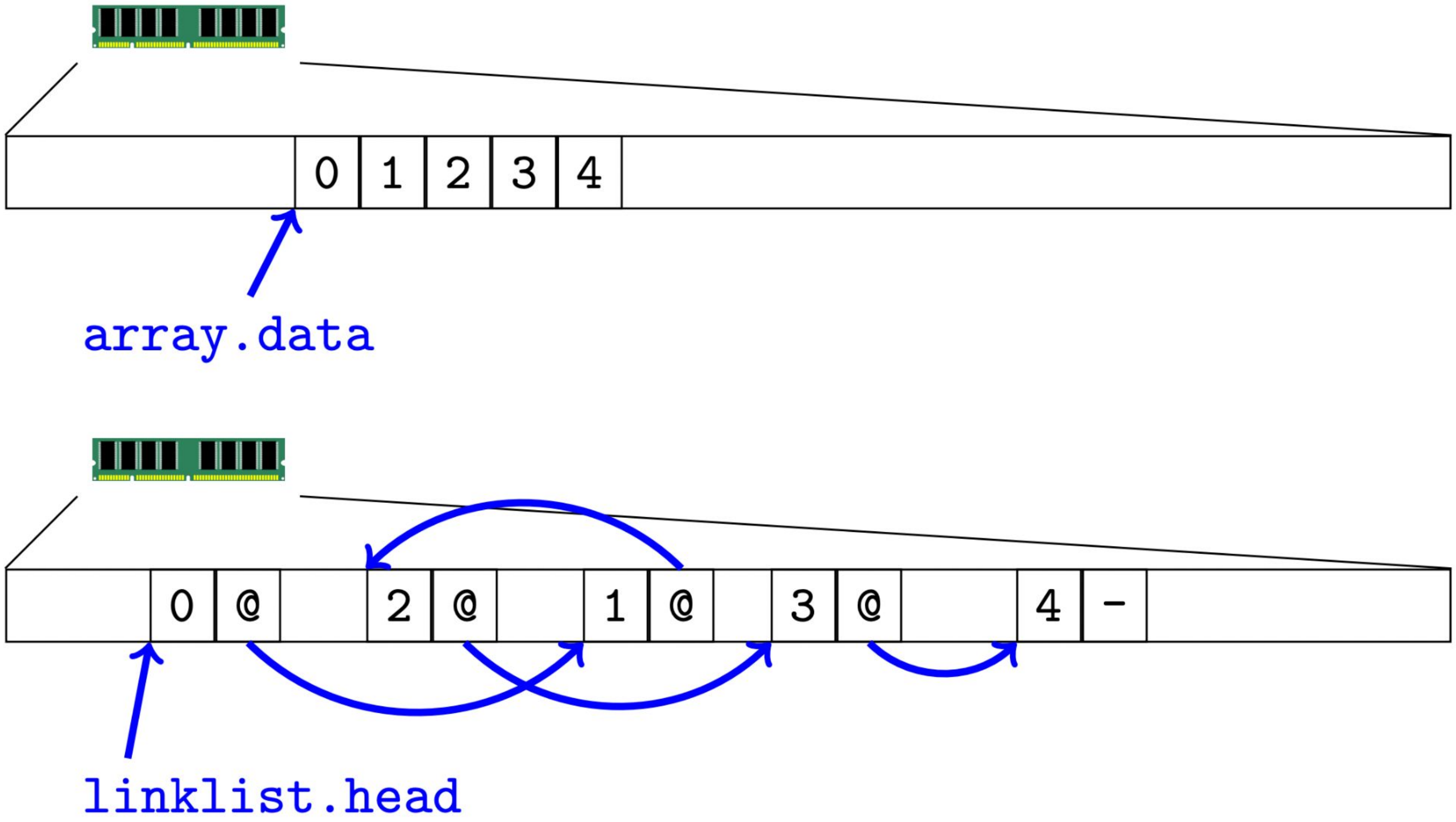
Different data structures will excel at different tasks

Collection ADTs

Property	Sequence	List	Set	Bag
Explicit Order	✓	✓		
Enforced Uniqueness			✓	
Fixed Size	✓			
Iterable	✓	✓	✓	✓

The Sequence ADT

```
public interface Sequence<E> {
  public E get(idx: Int);
  public void set(idx: Int, E value);
  public int size();
  public Iterator<E> iterator();
}
```



Arrays and Linked Lists in Memory

The List ADT

```java
public interface List<E>
    extends Sequence<E> { // Everything a sequence has, and...
  /** Extend the sequence with a new element at the end */
  public void add(E value);

  /** Extend the sequence by inserting a new element */
  public void add(int idx, E value);

  /** Remove the element at a given index */
  public void remove(int idx);
}
```

`List` Summary So Far (Lec 9,10,11)

	ArrayList	Linked List (by index)	Linked List (by reference)
`get(idx)`	$\Theta(1)$	$\Theta(\text{idx}) \subset O(n)$	$\Theta(1)$
`set(idx,v)`	$\Theta(1)$	$\Theta(\text{idx}) \subset O(n)$	$\Theta(1)$
`size()`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`add(v)`	$O(n)$, Amortized $\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`add(idx,v)`	$O(n)$	$\Theta(\text{idx}) \subset O(n)$	$\Theta(1)$
`remove(idx)`	$O(n)$	$\Theta(\text{idx}) \subset O(n)$	$\Theta(1)$

Sets

A **Set** is an **unordered** collection of **unique** elements.

(order doesn't matter, and at most one copy of each item)

The Set ADT

`void add(T element)`

Store one copy of **`element`** if not already present

`boolean contains(T element)`

Return true if **`element`** is present in the set

`boolean remove(T element)`

Remove **`element`** if present, or return false if not

Bags

A **Bag** is an **unordered** collection of **non-unique** elements.

(order doesn't matter, and multiple copies of the same item is OK)

The Bag ADT

`void add(T element)`

Store one copy of **`element`**

`int contains(T element)`

Return the number of copies of **`element`** in the bag

`boolean remove(T element)`

Remove one copy of **`element`** if present, or return false if not

Note: Sometimes referred to as multiset. Java does not have a native Bag/Multiset class.

Sets and Bags (Lec 11,15)

	LinkedList	ArrayList	ArrayList (sorted)
`Set.add`	$O(n)$	$O(n)$	$O(n)$
`Set.contains`	$O(n)$	$O(n)$	$O(\log(n))$
`Set.remove`	$O(n)$	$\Theta(n)$	$O(n)$
`Bag.add`	$O(1)$	$O(n)$, Amortized $\Theta(1)$	$O(n)$
`Bag.contains`	$\Theta(n)$	$\Theta(n)$	$O(n)$
`Bag.remove`	$O(n)$	$\Theta(n)$	$O(n)$

Misc

Code Analysis

You should be able to derive growth functions for a given piece of code (recursive and non-recursive!):

- Sequential (statements executed one after another)
  - Add the number of steps together
  - If I do **A** then **B**, the total cost is the **cost to do A** plus the **cost to do B**
- Selection (conditional execution of statements)
  - Our growth function will be a piecewise function
- Repetition (repeating execution of one or more statements)
  - Add up the total number of steps…with summations

Code Analysis (Lec 08)

```java
public void countDuplicates(Data[] data) {
  System.out.println("Counting duplicates");
  int count = 0;
  for (int i = 0; i < data.length; i++) {
    for (int j = i+1; j < data.length; j++) {
      if (data[i] == data[j]) {
        count++;
      }
    }
  }
}
```

$$T(n) = 1 + 1 + \sum_{i=0}^{n} \sum_{j=i+1}^{n} T_E(n, i, j)$$

$$T_E(n, i, j) = \begin{cases} 2 & \text{if data[i] == data[j]} \\ 1 & \text{otherwise} \end{cases}$$

Code Analysis (Lec 12)

```java
public int factorial(int n) {
    if(n <= 1) { return 1; }                          ← Base Case
    else { return n * factorial(n - 1); }  ← Recursive Case
}
```

$$T(n) = \begin{cases} \Theta(1) & \textbf{if } n \leq 1 \\ T(n-1) + \Theta(1) & \textbf{otherwise} \end{cases}$$

OR

$$T(n) = \begin{cases} c_0 & \textbf{if } n \leq 1 \\ T(n-1) + c_1 & \textbf{otherwise} \end{cases}$$

It's ok to write constants as $\Theta(1)$ or give them names

Accessing Data

We've now seen three different ways to access data:

- **By Index/Position:** get the value at a certain position
  - ie: give me the first value, last value, tenth value
  - Only applies to Lists/Sequences (Sets and Bags are unordered)
- **By Value:** find a particular value in a data structure
  - Part of the Set/Bag ADT (but could still be used with Lists/Seqs)
- **By Reference:** access via direct reference to part of the data structure
  - Can be used when you have a LinkedList data structure
  - It's not magic – you must have a way of getting/storing the reference

Sorting / Binary Search

MergeSort: Recursive sorting algorithm that sorts in $\boldsymbol{O}(\boldsymbol{n}\ \mathbf{log}(\boldsymbol{n}))$ time
(We'll see many other sorting algorithms this semester)

Why is knowing our data is sorted useful?
It can (potentially) make searching for a particular value faster!

Binary vs Linear Search (Lec 15)

Linear search:

- Removes one element from consideration each step, ***O*(*n*)**
- Does not require list to be sorted
- Does not require constant time random access

Binary search:

- Removes half of the elements from consideration each step, ***O*(log(*n*))**
- Requires list to be sorted
- **Requires constant time random access**
    - (binary search on a linked list is still linear time…)